Improving Instruction Supply Efficiency in Superscalar Architectures Using Instruction Trace Buffers

Chih-Po Wen

University of California at Berkeley

Best Available Copy

Abstract

This paper addresses the problem of efficient instruction supply in superscalar architectures. The presence of branch instructions introduces two types of overhead on instruction supply efficiency, namely the delay required to resolve branch decisions and the fetch penalty due to poor code alignment. The former can be reduced by conventional branch prediction techniques. The latter, however, has not been well addressed in past research. We propose the *instruction trace buffer* technique to alleviate the alignment problem. An instruction trace buffer is an aggressive extension of the conventional loop buffer technique. It caches recent instruction traces in a circular buffer to predict branch behavior as well as to improve code alignment. This approach relies on the fact that dynamic branch behavior is stable in most cases. Application traces are collected to verify this assumption and to evaluate our scheme. The result indicates that instruction trace buffers lead to substantial improvement on instruction supply efficiency for numerical applications. For these applications near perfect (99%) efficiency is achieved without imposing undue demand on the instruction memory bandwidth, which is essential in other hardware based alignment techniques.

Introduction

RISC processors have been successful in achieving an instruction issue rate of nearly one per clock cycle. To boost processor performance still further, superscalar architectures seem to be the next logical step. Similar to pipelined processors, data and control dependency are two major obstacles to substantial speedup. Dynamic scheduling techniques can be used to resolve data dependency. In these schemes the processor attempts to issue instructions continuously without waiting for the previous ones. The effectiveness of this approach is still limited by control dependency, since instruction supply must be halted until branch instructions are resolved. Therefore control dependency is really the major limiting factor for superscalar architectures [1] [2][3].

Branch prediction techniques are usually used to deal with control dependency. In these schemes the processor takes a guess whenever a branch instruction is encountered. Study has shown that we can actually come up with good guesses by remembering the past [4]. Hardware based branch prediction achieves a prediction rate of up to 90%, depending upon how much information is remembered. With some analysis and profiling, software-based branch prediction achieves roughly the same accuracy as dynamic, hardware based schemes [5]. However the responsiveness of software-based techniques is questionable: given the predicted branch outcome, the processor still has to wait for the correct target address. A complete software solution to this problem means a drastic change in the underlying instruction set, which is more or less against the spirit of superscalar architectures.

Good guesses only solve the first part of the problem. Even deterministic branches can do harm to fetch efficiency because they disturb the locality of instruction accesses. There is a penalty for discontiguous instruction fetches because the instruction memory is block structured. This is illustrated in Figure 1. In this scenario the instructions after the taken branch and those before the branch target have to be thrown away. As will be seen later in this paper, instructions thrown away in this fashion constitute a significant portion of the branch penalty.

Delay slot scheduling and compile time code layout may help solve the alignment problem. However, study has shown that it is very difficult to fill more than one delay slot [5]. For architectures with aggressive lookahead capability as superscalar processors, this is not sufficient. Moreover the number of delay slots is variable in our case, depending upon the alignment of branch instructions. This poses some code compatibility problems. The effectiveness of compile time code alignment seems to be limited, because restricting the layout of certain instructions may destroy code continuity in other cases (a good example is a code sequence containing two branch targets). A feasible solution is to have the fetcher align instructions for the decoders on the fly, which imposes extra bandwidth requirement for the instruction memory – the fetcher must fetch more instruction words than necessary, select the useful ones, and then merge them for the decoders. Here the gain may not justify the expense [2].

The alignment problem occurs mainly because instructions can not be aligned conveniently within a block-structured instruction memory, even if we could predict perfectly which direction they go at run time. To effectively deal with the alignment problem we propose the instruction trace buffer (TB) organization. The idea of the instruction trace buffer

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1992 ACM 0-89791-502-X/92/0002/0028...$1.50

is to cache recent instruction traces [1] in a circular buffer. Similar to a conventional loop buffer [7] [6], the TB starts operating when a backward branch into the buffer is detected. Subsequent instructions are delivered by the TB until a branch misprediction occurs. Instruction accesses in the trace buffer are always sequential regardless of the existence of PC-changing instructions. To handle long loops containing several instruction runs without demanding full associativity of the buffer, a small associative cache (called the *entry cache*) is used for loop detection. By compacting instructions into one sequential trace the TB achieves good code alignment; by delivering instructions according to previous branch outcomes it performs simple branch prediction.

The rest of the paper will be organized as follows. We first state the hardware assumptions and the application traces used for analysis throughout the paper. We then analyze the branch behavior using these traces and show the potential benefits of our approach. After giving the motivations we move on to the detailed mechanism and implementation of instruction trace buffers. A trace-driven simulator is written to evaluate our design and to compare our scheme with conventional instruction memory with branch prediction capability. Finally we discuss the simulation results and give our conclusions.

Hardware Assumptions and Application Traces

We will use a pipelined RISC processor as the backbone of our hypothetical superscalar architecture. The only pipeline stages of interest to us are the instruction fetch stage (IF) and the instruction decode (DE) stage. The structure of these two stages are very similar to those in the MIPS R2000 processor [8] or the Motorola 88100 processor [9].

Several Instructions are fetched into the processor in the IF stage. Because the way most instruction caches are structured, instruction fetches can not cross the *instruction block* [2] boundary. We take the number of decoders as the size of the instruction block. By the beginning of the next cycle these instructions are ready for decode in the ID stage. Branch conditions are calculated explicitly with *compare* instructions, and the results are stored in the genral purpose registers for later use. The hypotheticoal processor contains dedicated hardware for calculating the branch target address and the branch direction. Therefore the branch can be resolved before the end of the DE stage. The penalty for a mispredicted branch is one processor cycle. [3]

In addition we require that the processor have the ability to nullify invalid instructions in the pipeline when a mispredicted branch is detected. This assumption is true for all schemes involving *speculative* instruction fetches. However there is no danger of causing inconsistent register state, because none of the invalid instructions can execute beyond the DE stage. Detailed timing assumptions are given in Figure 2.

[1]an instruction run is defined as the sequentially fetched instructions between taken branches, whereas an instruction trace is the sequentially "executed" codes ended with a backward branch to the start of the trace.

[2]this is different from a cache block, which may span several instruction blocks.

[3]it is possible that the processor need more than one cycle to access the register file and to calculate the target address. However this does not affect our design.

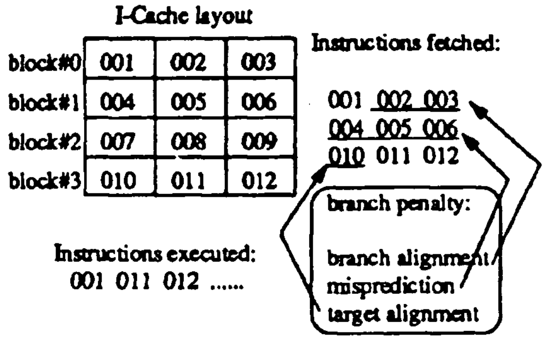


Figure 1: Branch Penalty Examples

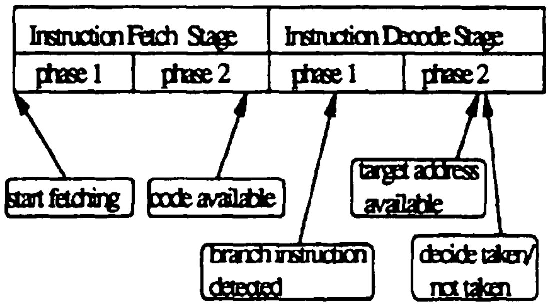


Figure 2: Instruction Fetch and Decode Timing

The application traces are collected on a MIPS machine using the *pixie* trace facility. Inputs are drawn from the sample data files associated with the benchmark programs. Programs were run only long enough to suppress the start-up effect (this is observed when the statistics starts to stablize). Table 1 describes the application traces used in this paper:

Dynamic Branch Behavior Measurements

The ultimate goal of instruction trace buffers is to optimize instruction fetches for loops with modest size, which is the common case claimed by the well known 90/10 rule. To give solid proof to the validty of our approach we ran a series of benchmark programs and obtained the measurements shown in Figure 3. We call an loop iteration *stable* if the instruction sequence executed therein is exactly the same as the one in the previous iteration. We call an loop iteration *fresh* if it is not stable. From these results we come up with the following observations:

- Over 99% of the instructions are delivered by loops with sizes smaller than 1024 instruction words (except for the dinero benchmark). This means we can achieve substantial gain if we optimize for these loops. The result also shows that we can capture most of the stable loops with fewer than 128 instructions.

- In average around 50% of the instructions are delivered by stable iterations. That is, recent instruction traces can be reused 50% of the time. This gives an estimate of how

program	description	code size (bytes)	trace size (instructions)
dhrystone	numerical benchmark	41360	2000000
doduc	Monte Carlo simulation	386800	1000000
dinero	cache simulator	73480	2000000
matrix300	300x300 matrix multiplication	219904	1000000

Table 1: Benchmark programs

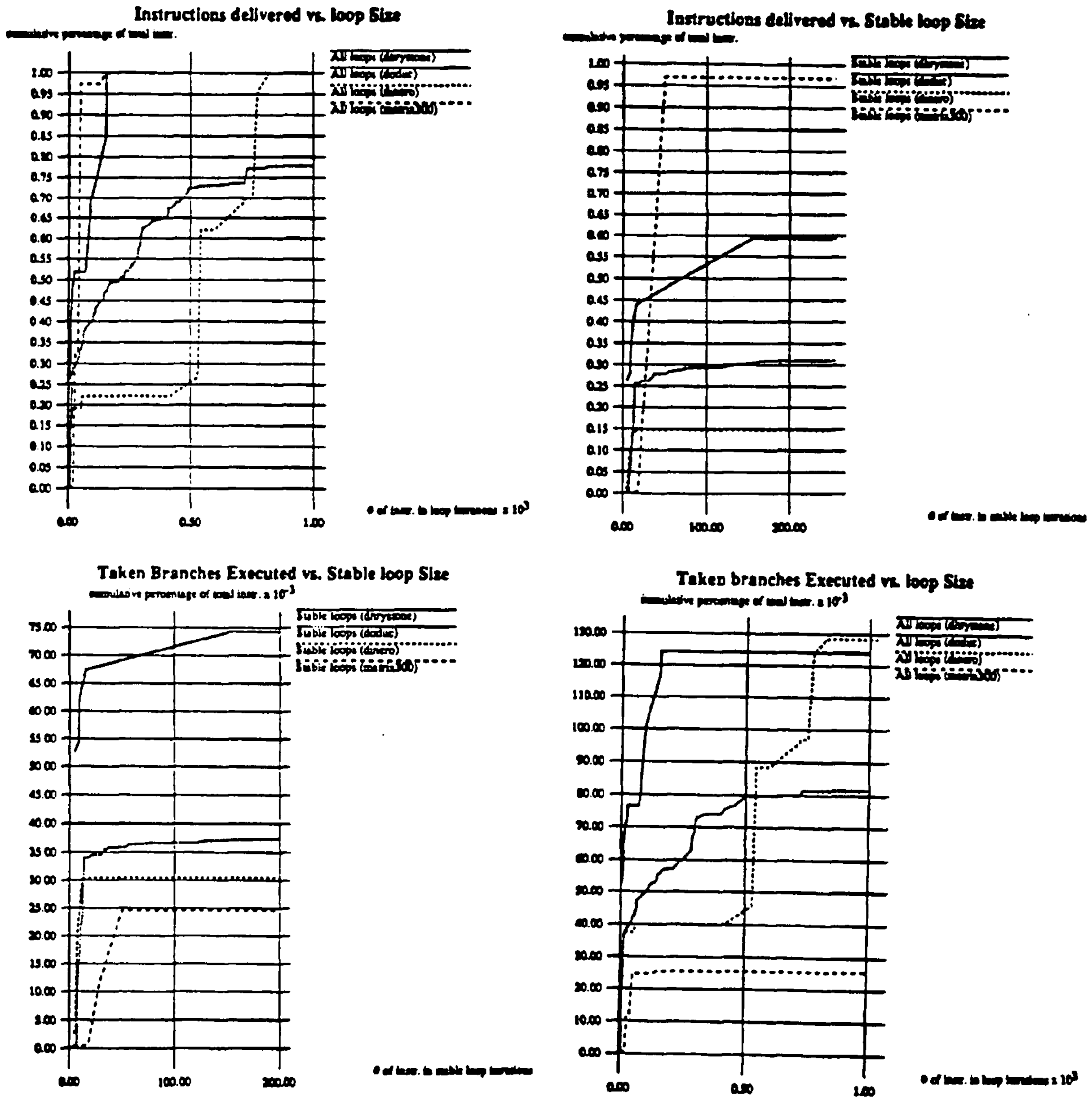


Figure 3: Dynamic Branch Behavior

often the TB would be activated at run time. However this ratio varies with different application programs.

- A large portion of the codes (50% in average) are still delivered by fresh iterations. These instructions evade the TB optimization and must be handle by conventional branch prediction techniques.
- 2–13% of the dynamic instruction count is constributed by taken branches. Above 50% of these are from stable iterations. This may not seem to be a major plus for the TB technique at first glance. However *each* of these taken branches is going to translate into *several* instructions of alignment penalty when the TB is not used. This indicates that the saving achieved by a TB can be substantial.

From the above statistics it is reasonable to conclude that the TB is not itself a complete solution. However it can be used to boost the performance of conventioal branch prediction techniques. In the next section we describe the detailed mechanism of instruction trace buffers.

Mechanism of Instruction Trace Buffers

The idea of TB is motivated by the fact that most codes appear to be more predictable than they appear. For example, consider the following C program that performs an in-order traveral of a binary tree. The code sequence executed at run time is also shown (read from the leftmost column to the rightmost column).

```
** program **              ** code executed **
    int inorder(p)               1    .    .
     btree *p;                   3    .    .
    {                            1    1    1
1:   if (p->data==NULL)          3    2    3
2:      return;                  1    4    1
3:   inorder(p->left);           3    5    3
4:      printf("%d",p->data);    .    1    .
5:   inorder(p->right);          .    3    .
6:   return;                     .    .    .
    }
```

The program behaves like a loop with an if-then constructs, with one branch of the if-statement taken more frequently than the other. If no optimization is done we have to pay for the same branch overhead again and again. However, if we could somehow cache the "preferred" loop iteration (containing statement 1 and 3 in this example), the saving can be significant. Now we present the structure of a TB and the mechanism required to achieve this goal.

Instruction Trace Buffer Organization

The organziation of a TB is presented in Figure 4. It consists of a circular *instruction buffer* and an *entry cache*. The instruction buffer holds the trace instructions, the predicted branch directions, and the predicted target addresses. The last item is required because the target address of a branch instruction may change over time (which is typical for subroutine returns). The entry cache stores a list of address/pointer pairs. The address part denotes the location of a backward branch, and the pointer gives the position of the branch target in the

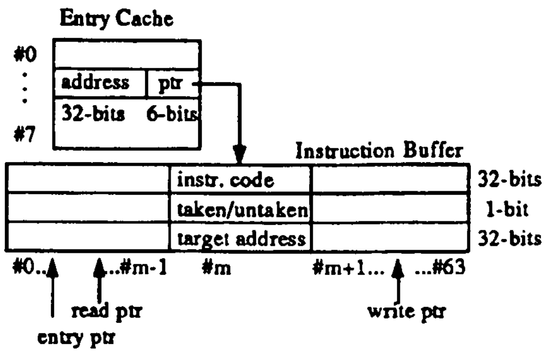


Figure 4: Organization of a Trace Buffer
The figure assumes an 8-slot entry cache and a 64-entries instuction buffer. The instruction buffer behaves like a FIFO buffer during trace collection. During trace emission the entry pointer and the write pointer define the boundary of the cached loop iteration.

instruction buffer. Looping is detected when the program is about to jump to first instruction of the loop iteration cached in the TB. An example of trace storage in a TB is given in figure 5.

The use of an entry cache is to lift the associativity requirement in conventional loop buffers. Since the size of a TB must be large enough to hold instruction traces, we can not afford to have a fully associative TB. The cost of implementing such a TB and the time required to access it will make the approach infeasible. Therefore we choose to separate loop detection from loop storage. This implies that only several instructions in the trace can be designated as loop entries. Our simulation results indicate that this is not a serious limitation. In our simulations we use an entry cache containing fewer than 8 entries, and we observe no substantial cache replacement activities – most of the entries are invalidated before they can be replaced due to the limited instruction buffer size. For the same reason a simple FIFO replacement policy suffices.

The instruction buffer does not require full addressing generality. All read and write accesses to it are sequential. Therefore we choose to organize the instruction buffer into several banks, with the number of banks equal to the number of decoders in the processor. Since the reads and writes may start with any position in the buffer, we need extra logic to route the correct data item to/from the various banks. Figure 6 and Figure 7 suggest an implementation of the bank address generation logic and the data route logic.

The speed requirement of a TB is roughly the same as that of a register file in RISC processors – each basic operation (defined below) must be completed in half an cycle. Although the size of the instruction buffer is usually larger than that of a register file, we believe the speed requirement can be satisfied by the proposed implementation.

The basic operations on the TB is summarized below:

- WRITE: write a sequence of instructions to the buffer, starting from the position indicated in the write pointer.

```
Instruction addresses in the trace:
  100  101  110  111  112  120  121  122  123  124  110 ...
        -               -                       -
      taken branch   taken branch            backward branch

Instruction buffer content:
                     #0 #1 #2 #3 #4 #5 #6 #7 #8 #9 #10 #11
  instruction code:        <omitted>
  taken/untaken:      0  1  0  0  1  0  0  0  0  1
  target address:     - 110 -  - 120 -  -  -  - 110

Write pointer (designating the end of the loop iteration): 10
Entry pointer (designating the start of the loop iteration): 2
Entry cache content:    {  (124,2)  }
```

Figure 5: Trace storage example.

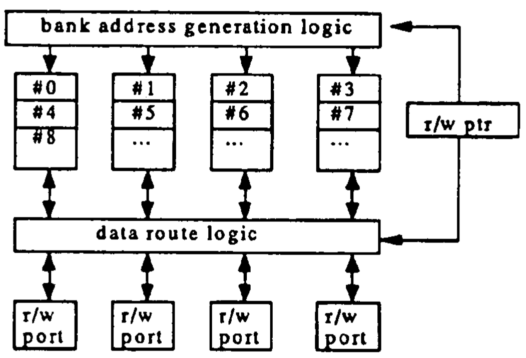


Figure 6: Implementation of a Trace Buffer
The implementation given in this figure is meant for a 4-decoder superscalar processor. The instruction buffer is organized into 4 banks, each with its own address generation and data route logic. Simultaneous accesses to the instruction buffer will always be directed to different banks because of their sequentiality.

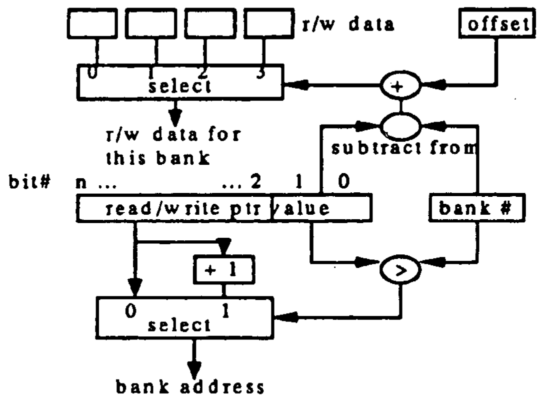


Figure 7: a slice of the address generation logic and the data route logic
The data fed in from the r/w ports are multiplexed to the appropriate banks. the *offset* value designates the port number of the first valid instruction, whose buffer address is given by the read/write pointer. The selected data item is then read from or written to the bank address.

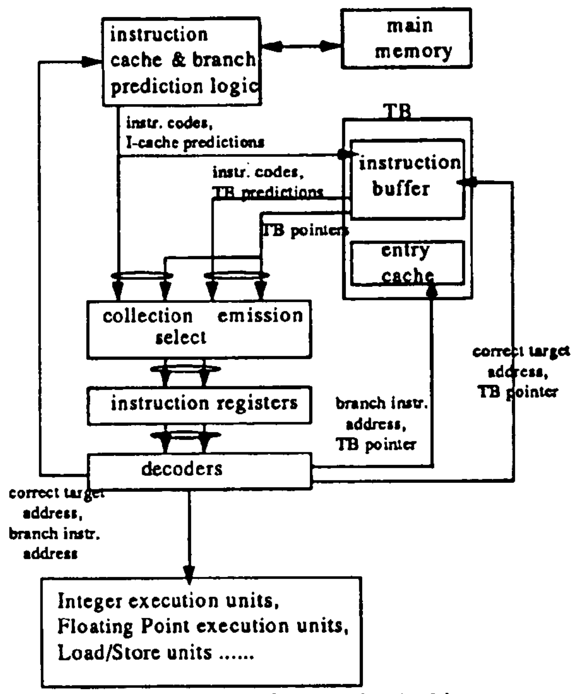
These instructions and the associated prediction information are supplied by the instruction cache. Note that the instruction buffer is maintained in a circular fashion.

- READ: read a sequence of instructions from the buffer, starting from the position indicated in the read pointer. If the read has gone past the last instruction of the current loop iteration, wrap around to the beginning of the loop.
- ADVWPTR: advance the write pointer. Also invalidate stale data in the entry cache
- ADVRPTR: advance the read pointer.
- LOOKUP: look up an address in the entry cache.
- ADD: add an address/pointer pair to the entry cache.
- ROLLBACK: rollback the write pointer when a misprediction occurs.
- SETPTRS: adjust the positions of the read and entry pointers when the program enters a trace
- CORRECT: correct the prediction related information in the buffer. This happens when a branch is mispredicted.

To ensure uninterrupted instruction supply some precaution must be taken. Bank collisions may occur when we attempt to deliver instructions from different repetitions of the loop iteration. This is because the buffer addresses of these instructions are not contiguous. We solve this problem by appending the first several instructions to the end of the loop iteration. This guarantees that buffer accesses are always contiguous and thus no bank conflict can occur. Care must be taken when we wrap around the read pointer so that the instructions appended will not be redelivered.

Integrating the TB and the Superscalar Processor

In this section we discuss the interaction between the TB and other components in a supersalar architecture. The block diagram of an integrated superscalar architecture is shown in Figure 8. The processor contains an instruction cache (possibly equipped with branch prediction hardware), a set of decoders (four in this example), the execution units, and the TB. Aside from the instruction registers extra buffers are required to hold the the branch prediction information and


instruction cache & branch prediction logic
main memory
instr. codes, I-cache predictions
TB
instruction buffer
instr. codes, TB predictions
TB pointers
entry cache
collection emission select
instruction registers
branch instr. address, TB pointer
correct target address, TB pointer
decoders
correct target address, branch instr. address
Integer execution units, Floating Point execution units, Load/Store units ......
Figure 8: Proposed Superscalar Architecture

Three items are always passed into the pipeline. They are the instruction code, the predicted target address (including the predicted direction), and the position of the instruction in the TB. During trace collection the I-cache supplies the former two for each instruction, with the third information tagged by the TB. During trace emission the TB supplies all three items. Feedback from the decoders are sent to the I-cache as well as to the TB to update branch prediction data.

the TB pointers in the pipeline. These are used for detecting mispredicted branches and for managing the TB. Their use will become clear later.

At any instant the processor can be either in the *trace collection* mode or in the *trace emission* mode. During trace collection instructions are fetched from the instruction cache and the TB simply records the instructions in the instruction buffer. Each instruction deliverd by the instruction cache is tagged with its position in the TB. When the instructions are decoded the entry cache attempts to look up the first backward branch instruction[4] in the fetched block. The processor must also detect the following three events:

1. A branch is mispredicted by the instruction cache: this is done by comparing the resolved target address with the predicted target address. When this happens the processor invalidates the instructions fetched in the IF stage, change the PC accordingly, and update the instruction buffer content.
2. A backward branch missed by the entry cache: the address of the backward branch and the next position to write in the TB are added to the entry cache. Using this information we can detect a second visit to cache loop later.

[4]a backward branch is defined in a limited context. They are meant to be conditional branch instructions using PC-relative address mode. The offset, the, must be negative.

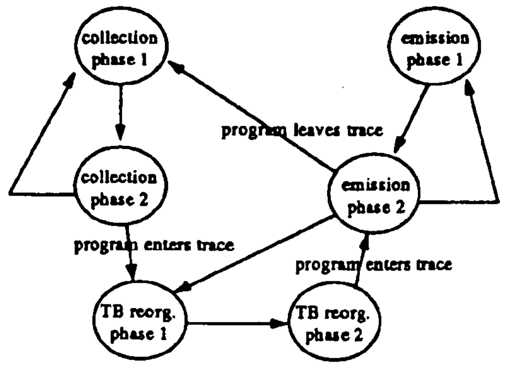


Figure 9: State Transition Diagram of the TB controller

3. A backward branch is correctly predicted by the entry cache: we say the program now *enters the trace*, and subsequent instructions can be delivered from the TB. Now the processor switches to the trace emission mode.

During trace emission instructions are supplied from the TB. Perfect efficiency (that is, no branch penalty) can be sustained when the processor is executing in this mode. The TB also tags the instructions with the prediction information before sending them to the instruction registers. Now the processor has to watch for the following two events:

1. A branch is mispredicted by the TB: the program is said to have *left the trace*. Similarly we discard invalid instructions in the pipeline and update the content of the instruction buffer. The processor now switches to the trace collection mode. Trace collection by the TB starts from where the program left the trace.
2. A backward branch is correctly predicted by the entry cache: the program enters a nested loop. The processor updates the content of the TB and enters the inner loop.

The content of the TB must be reorganized to avoid bank conflicts during the transition from the collection mode to the emission mode. The process is explained in the previous section. The state transitions are illustrated in Figure 9. The state operations are also summarized in Table 2.

Performance Evaluation

All results in this section are expressed in terms of instruction fetch efficiency. The instruction fetch efficiency is defined as the ratio of executed instructions and the instructions delivered to the instruction decode stage in the pipeline. The ratio is 1 for prefect efficiency where no instructions are discareded.

Three superscalar processor configurations are simulated. The first configuration (referred to as C0) uses a standard instruction cache and employs no branch prediction scheme. The second configuration (referred to as CB) augments the first with an branch target buffer (BTB). The BTB takes the *address* of a branch instruction and returns a guess of the *target address*. The third configuration (referred to as CTxx, where xx is the number of entries in the instruction buffer) adds a TB on top of the I-cache and the BTB.

collection phase 1	I-cache Fetch
	LOOKUP first backward branch being decoded
	CORRECT instruction buffer if flag set
5	next state is collection phase 2
collection phase 2	if (there is a mispredicted branch)
	flush invalid instructions in the pipeline
	update I-cache prediction info.
	BACKOFF to where the misprediction occurs
	save correct branch info. and set flag
	else
	WRITE fetched instruction block to TB
	ADVWPTR
	if (first backward branch is taken and not in entry cache)
	ADD its address and TB position to the entry cache
	if (first backward branch is taken and in entry cache)
	next state is TB reorg. phase 1
	else
	next state is collection phase 1
emission phase 1	READ instructions from TB
	ADVRPTR
	LOOKUP first backward branch being decoded
6	next state is emission phase 2
emission phase 2	CORRECT instruction buffer if flag set
	if (there is a mispredicted branch)
	flush invalid instruction sin the pipeline
	BACKOFF to where the misprediction occurs
	save correct branch info. and set flag
	else
	ADVRPTR
	if (first backward branch is taken and not in entry cache)
	ADD its address and TB position to the entry cache
	next state is collection phase 1
	if (first backward branch is taken and in entry cache)
	next state is TB reorg. phase 1
	else if (no mispredicted branch)
	next state is emission phase 1
	else
	next state is collection phase 1
TB reorg. phase 1	SETPTRS
	READ first block of instructions from TB
	send them to the instruction registers
	next state is TB reorg. phase 2
TB reorg. phase 2	WRITE them after where the misprediction occurs
	ADVRPTR
	ADVWPTR
	next state is emission phase 1

Table 2: State operations of a TB

For convenience of discussion we classify the branch penalty into three types. The *misprediction penalty* is incurred when the instruction block fetched is not the correct one. The *branch alignment* penalty is incurred when a taken branch is not the last in the instruction block. Finally the *target alignment* penalty occurs when the branch target is not the first in the instruction block. Therefore we expect CB to eliminate portions of of misprediction penalty, and CT to reduce the branch and target alignment penalty.

Results

To avoid explosion in the parameter space we fix the size of the BTB entries (128 entries using one prediction bit) and the entry cache (8 entries). The choice of BTB size is based on the result of the [4] study. A 128-entry BTB using one prediction bit yields a prediction rate of 82%. The prediction rate can be increase to 88% by using 16 times as many entries, but that is immaterial to our purpose here. The size of the entry cache is also not important as we observe no significant replacement activities in our simulation – space in the entry cache is usually available because entries are frequently invalidated when new data are written into the instruction buffer. For the CT configuration we simulate 3 different instruction buffer sizes (32,64,128). The results are given in Table 3 and Table 4.

Program	C0	CB	CT32	CT64	CT128
dhrystone	41%	62%	99%	99%	99%
doduc	60%	72%	78%	79%	80%
dinero	49%	68%	71%	71%	71%
matrix300	85%	93%	99%	99%	99%

Table 3: Comparision of Instruction Fetch Efficiency

It can be seen that branch misprediction penalty in CB is small (1–9%) when compared with the alignment penalty (6–60% without TB). Therefore the net effect of using a BTB is to shift the branch bottleneck from the misprediction penalty to the alignment penalty. The fetch efficiency without any prediction scheme is simply unacceptable.

The simulation results also show that instruction trace buffers lead to significant improvement for numerical computation programs. The dhrystone and matrix300 programs fit well into this category. For programs with small branches and loops such as the doduc benchmark, a great portion of the alignment penalty can be reduced by a TB. However the TB slightly increases the misprediction penalty because it attempts to predict several branches as a whole, and extra penalty is incurred when we enter and leave the trace too often. For programs whose behavior depends heavily on the inputs the effectiveness of TB is limited. This is indicated in the results of the dinero benchmark. In this case the TB is activated only 10% of the time. The size of the TB does not seem to be an important factor, because a 32-entry TB shows

Program	C0	CB	CT32	CT64	CT128
dhrystone	80%	1%	1%	1%	1%
doduc	40%	9%	10%	10%	10%
dinero	70%	8%	8%	8%	8%
matrix300	11%	1%	1%	1%	1%

Mispediction penalty

Program	C0	CB	CT32	CT64	CT128
dhrystone	60%	60%	1%	1%	1%
doduc	30%	30%	16%	15%	15%
dinero	39%	39%	33%	33%	33%
matrix300	6%	6%	1%	1%	1%

Alignment penalty

Table 4: Comparision of branch penalty. The results are in terms of the penalty per executed instruction.

indentical results as a 128-entry TB in almost all cases.

To deal with general purpose programs more effectively we suggest the following improvement:

- Our implementation of TB tends to optimize only the innermost loops. This phenomenon can be observed when the program leaves the cached trace. At this point no attempt is made to recapture the instruction trace of an outer loop. We believe this problem can be dealt with with a more sophisticated TB control algorithm, albeit at the cost of greater complexity.
- The TB mechanism presented in this paper is a pure hardware solution and it seems wasteful not to utilize the knowledge available at compile time. For example, two iterations are required to identify and store a loop in the TB – the first iteration only serves to locate the backward branch. If the compiler can tag the loop entries (possibly by inserting a dummy instructions) with the corresponding backward branches, this overhead can be saved. A better idea will be to utilize the language semantics to tell which loops are suitable for optimization. For example, *if-then* statements make good candidates for the TB optmization, while *switch* statements usually are not.

Concluding Remarks

In this paper we present the trace buffer technique for optimizing the instruction fetch performance. Simulation results show great improvement for numerical applications, using a TB containing only 32 entries. On the other hand, the gain with general purpose applications is limited. To boost TB performance further, we suggest that compile-time knowledge be used to selectively optimize the loops in a program.

Our discussion so far has been limited to the instruction fetch performance. Whether this will lead to overall performance improvement is not investagated. However we believe that parallelism can be increased when we overlap different in-

[5]this is for correcting the branch mispredicted in the *previous* cycle. The CORRECT operation implies a write in the instruction buffer. Therefore we have to delay it to the next cycle to avoid conflicts with the normal WRITE operation

[6]this step is skipped if the backward branch is the last instruction in the current loop iteration

struction runs. The combined effect of instruction fetch efficiency and data dependency seems to be worthy of further investigation.

Acknowledgements

The author would like to thank Tzi-cker Chiueh for his valuable comments and his help with the collection of application traces. The author also would like to thank C. V. Ramamoorthy for providing the resources needed to complete this work.

References

[1] N. P. Jouppi and D. W. Wall, *Available Instruction-Level Parallelism for Superscalar and Superpiplined Machines, Proc. Third Conf. ASPLOS*, IEEE/ACM (April 1989), Boston, Mass, pp. 290-302.

[2] M. Johnson, *Superscalar Micorprocessor Design*, Englewood Cliffs, NJ:Prentice-Hall, 1991.

[3] M. D. Smith, M. Johnson and M. A. Horowitz, *Limits on Multiple Instructions Issue, Proc. Third Conf. ASPLOS*, IEEE/ACM (April 1989), Boston, Mass, pp. 290-302.

[4] C. H. Perleberg, *Branch Target Buffer Design, Report No.UCB/CSD 89/553*, Computer Science Division, University of California, Berkeley.

[5] S. McFarling and J. Hennessy, *Reducing the Cost of Branches, Proc. 13th Symposium ISCA*, June 1986, pp. 396-404.

[6] J. E. Thorton, *Design of a Computer – The Control Data 6600* , Glenview IL:scott, Foresman and Co., 1970.

[7] R. M. Tomasulo, *An Efficient Algorithm for Exploiting Multip le Arithmetic Units*, IBM Journal, Vol. 11 (Jan. 1967), pp. 25-33.

[8] G. Kane, *MIPS R2000 RISC Architecture*, Englewood Cliffs, NJ :Prentice-Hall, 1987.

[9] Motorola, *Inc. MC88100 RISC Processor User's Manual*, Englewood Cliffs, NJ:Prentice-Hall, 1989.

This Page is Inserted by IFW Indexing and Scanning Operations and is not part of the Official Record

BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images include but are not limited to the items checked:

- ☐ BLACK BORDERS
- ☐ IMAGE CUT OFF AT TOP, BOTTOM OR SIDES
- ☐ FADED TEXT OR DRAWING
- ✓ BLURRED OR ILLEGIBLE TEXT OR DRAWING
- ☐ SKEWED/SLANTED IMAGES
- ☐ COLOR OR BLACK AND WHITE PHOTOGRAPHS
- ☐ GRAY SCALE DOCUMENTS
- ✓ LINES OR MARKS ON ORIGINAL DOCUMENT
- ☐ REFERENCE(S) OR EXHIBIT(S) SUBMITTED ARE POOR QUALITY
- ☐ OTHER: ______________________

IMAGES ARE BEST AVAILABLE COPY.
As rescanning these documents will not correct the image problems checked, please do not report these problems to the IFW Image Problem Mailbox.